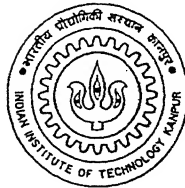


Longest Common Subsequence - A new Data Structure and a Related Problem

*A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of
Master of Technology*

by

Gnanavardhan R H



to the

Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur

July, 2005

Certificate

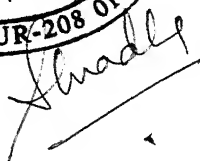
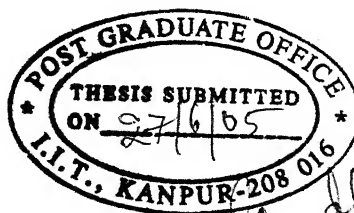
This is to certify that the work contained in the thesis entitled "*Longest Common Subsequence - A new Data Structure and a Related Problem*", by *Gnanavardhan R H*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.



June, 2005

(Dr. Sanjeev Saxena)

Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.



TU

CSE/2005/M

G5312

19 3 OCT 2005

पुस्तकालय केवलकर पुस्तकालय

भारतीय संस्थान कानपुर

पुस्तक संख्या 153073



A153073

Abstract

In this thesis we propose an efficient algorithm for solving the Longest Tandem Subsequence(*LTS*) problem. The earlier algorithm (A. Kosowski, *SPIRE(2004)*, 93-100) for finding the *LTS* of a sequence $S[s_1s_2...s_n]$ takes $O(n^2)$ space and time, where ‘ n ’ is the size of the sequence. Our algorithm takes $O(n\sigma + dp \log d)$ time and $O(n\sigma + np)$ space. Here ‘ σ ’ is the size of the alphabet, ‘ p ’ is the size of the output (length of the *LTS*), and $(d = n - 2p + 1)$.

We also describe a new data structure for finding the Longest Common Subsequence (*LCS*) between two sequences $D[d_1d_2...d_n]$ and $Q[q_1q_2...q_m]$, $(n \gg m)$. The data structure requires $O(n)$ time and space for construction, assuming that the size of the alphabet, σ is “large” ($\sigma \geq (n/\log^2 n)$). The time complexity of our *LCS* algorithm is $O(mp \log \log n)$, where ‘ p ’ is the length of the *LCS*.

Finally, we describe some variations of the search data structure used in solving the *LCS* problem. These variations allow us to specify any search parameter λ (where $\lambda = \Omega(1)$ and $\lambda = O(\log n)$) with trade-offs between space and time required for constructing the data structure.

Acknowledgements

I offer this work to that Supreme Force, which is the very reason for an idea to strike one's mind.

I would like to express my earnest gratitude towards my thesis supervisor, Dr. Sanjeev Saxena for guiding me through the course of my thesis. His guidance at every level – starting from searching for a journal in the library to the point of writing a good technical report – has been invaluable.

I would like to thank all of CSE Faculty for providing me with an opportunity to study here and also for nurturing my academic interests.

I thank all my friends of M.Tech 2003, particularly Arindam, Ashish, Chaitanya, Ganga, G Rajesh, Janardhan, Madhav, Pratik, Rahul, Satyaki and Surya for their assistance with my thesis.

I thank my parents for the countless sacrifices they have made to bring me up to this stage in life and my brother Srihari, to whom I owe all my achievements till now.

I am grateful to the Kannada Sangha at IITK, being with them is like being a part of a big family at a home away from home.

Finally, I wish to thank the IITK community for making IITK such a great place, that it has always been.

Contents

1	Introduction	1
1.1	Organization of the Thesis	2
2	Longest Common Subsequence (<i>LCS</i>)	3
3	Longest Tandem Subsequence(<i>LTS</i>)	6
3.1	Previous Work	6
3.1.1	Algorithm: <i>LTSsplit</i>	7
3.2	Our Approach	8
4	The Algorithm	10
4.1	Preprocessing	10
4.2	<i>Re-adjust Contours</i>	11
4.3	<i>Extend Contours</i>	13
4.4	Shifting in the Reverse Direction	14
4.5	Computing the current <i>LCS</i>	15
5	Correctness and Analysis	16
5.1	Correctness of the Algorithm	16
5.2	Analysis	16
6	A New Data Structure for the <i>LCS</i> Problem	19
6.1	Introduction	19
6.2	Related Work	20
6.3	Preliminaries	21
6.4	The Document Search Tree (<i>DST</i>)	21

6.4.1	Construction	21
6.4.2	Searching the <i>DST</i>	23
6.5	The Algorithm	24
6.5.1	Constructing the Contours	25
6.5.2	Detecting the Dominant Matches	25
6.6	Analysis	26
6.6.1	Preprocessing	26
6.6.2	The Algorithm	27
6.7	Comparison with the existing algorithms	27
7	Different Search Methods	29
7.1	Introduction	29
7.2	Different Search Methods	29
7.2.1	Search Cost $\lambda = O(1)$	30
7.2.2	Search Cost $\lambda = O(\log n)$	30
7.2.3	Search Cost $k\lambda$, <i>Levels</i> of <i>DST</i> = k	30
7.2.4	Search Cost = λ , Unrestricted number of <i>Levels</i>	32
7.2.5	Selective Representation	32
7.3	Comparison of the Different Search Methods	33
8	Conclusions	34
A	The Algorithm(Pseudocode)	35
B	Splay Trees	38
C	Comparison Results	39
	Bibliography	43

List of Tables

6.1	Previous work on <i>LCS</i>	20
6.2	Comparative Analysis of <i>LCS</i> algorithms($\sigma \geq (n/\log^2 n)$)	28
7.1	Comparison of the Different Search Methods	33
C.1	Comparative Analysis of <i>LCS</i> algorithms($\sigma \geq (n/\log^2 n)$)(Time in μs) . . .	39

List of Figures

2.1	Contours and Dominating Matches in $m \times n$ matrix M	4
3.1	Contour Modification (after shifting a from S_R to S_L)	9
4.1	<i>Re-adjust Contours</i> (after removing a from S_R)	12
6.1	DST for the sequence $D = aabbcdbccccddbdd$	22
6.2	Type-I and Type-II Lists for symbol a	23
7.1	Type-I and Type-II Lists for symbol a	30
C.1	$\sigma = 40$	40
C.2	$\sigma = 50$	40
C.3	$\sigma = 65$	41
C.4	$\sigma = 80$	41
C.5	$\sigma = 94$	42
C.6	$\sigma = 94$ and $n \gg m$	42

Chapter 1

Introduction

A character sequence S of length n over a nonempty alphabet Σ is a function $S : \langle 1, n \rangle \longrightarrow \Sigma$ (here $\langle i, j \rangle$ is used to denote the set of consecutive numbers $\{i, i+1, \dots, j\}$). The symbol s_i , where $1 \leq i \leq n$, is used to denote the i^{th} element of the sequence S . The size of the alphabet is denoted by σ , that is $|\Sigma| = \sigma$.

Example $S = aabccabbc$, $\Sigma = \{a, b, c\}$ is a sequence of size 9, where we have $s_1 = a$, $s_2 = a$, $s_3 = b$... and $s_9 = c$. ■

Definition 1.1 A sequence S of length n is called a *tandem* sequence if n is an even number and $\forall_{1 \leq i \leq n/2} s_i = s_{i+n/2}$.

Example $S = abcaabca$, is a *tandem* sequence of length 8 with $s_1 = s_5$, $s_2 = s_6$, $s_3 = s_7$ and $s_4 = s_8$. ■

Definition 1.2 A sequence T , $|T| = m$, is called a *subsequence* of S $|S| = n$, if it is possible to define an increasing function $h : \langle 1, k \rangle \longrightarrow \langle 1, n \rangle$, such that $\forall_{1 \leq i \leq k} t_i = s_{h(i)}$. This relation between the sequences T and S is written in the form $T \subseteq S$.

Example $T = acba$ is a subsequence of the the sequence $S = abcabca$, where $t_1 = s_1$, $t_2 = s_3$, $t_3 = s_5$ and $t_4 = s_7$. ■

The Longest Tandem Subsequence (*LTS*) problem for a given sequence S is the problem of finding a tandem sequence T such that $T \subseteq S$ and the length of the sequence T is the maximum possible.

Example The longest tandem subsequence of the sequence $S = \underline{a}db\underline{a}cc\underline{a}cb\underline{d}c$, is the sequence $T = abcabc$ of length 6 with $s_1 = s_7, s_3 = s_9$ and $s_5 = s_{11}$. ■

The Longest Common Subsequence $LCS(A, B)$ of two sequences A and B is a sequence P of the maximum possible length such that $P \subseteq A$ and $P \subseteq B$.

Example The longest common subsequences between the sequences $A = aabcaacb$ and $B = bbbcabcb$ are $P = \{bcac, bcab\}$. ■

1.1 Organization of the Thesis

The thesis is organized as follows. In Chapter 2 we discuss some preliminaries related to the Longest Common Subsequence problem. In Chapter 3 we talk about previous and some related work to the Longest Tandem Subsequence problem. We also briefly describe our approach in this chapter. Then, in Chapter 4 we describe our algorithm in detail. The correctness of the algorithm and analysis are covered in Chapter 5. In Chapter 6 we propose a new algorithm to the Longest Common Subsequence problem. Finally, in Chapter 7 we give some variations of the data structure proposed in Chapter 6.

Chapter 2

Longest Common Subsequence (*LCS*)

Given two sequences $A = a_1a_2...a_m$ and $B = b_1b_2...b_n$, $m \leq n$, over some alphabet Σ of size σ , the longest common subsequence (*LCS*) problem is to find a subsequence of greatest possible length that occurs in both A and B .

Next, we define a few concepts related to the *LCS* problem, as presented in Rick's ¹ [14] work. These concepts were first introduced by Hirschberg[5].

Definition 2.1 A *match* is an ordered pair (i, j) , $1 \leq i \leq m$ and $1 \leq j \leq n$, such that $a_i = b_j$.

The set of all matches between A and B can be represented in a $m \times n$ matrix, M . Two matches (i, j) and (i', j') may be part of a common subsequence if and only if $(i < i' \wedge j < j')$ or $(i' < i \wedge j' < j)$.

Definition 2.2 A *chain* is defined as a sequence of matches that is increasing in both the components i.e. in terms of both i and j (indicated by the dotted line in Fig. 2.1).

The longest common subsequence between two sequences A and B can be now seen as the longest chain of such matches that are strictly increasing in both the components.

A match (i, j) is said to have a rank k if the length of the longest chain ending at (i, j) is k . The set of matches in M can be partitioned into classes $C_1, C_2...C_p$, each of which

¹Verbatim from [14]

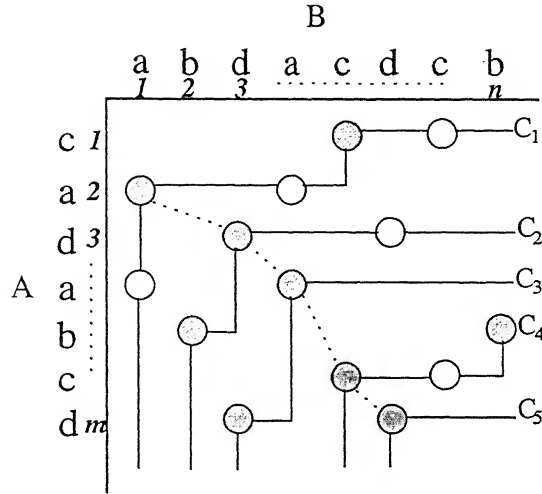


Figure 2.1: Contours and Dominating Matches in $m \times n$ matrix M

contain matches of the same rank. The matches belonging to the same class can be ordered in the form of a *contour* (along increasing order of the second component j) as shown in Fig. 2.1.

Lemma 2.1 *Contours of different class never cross each other or touch each other.*

Proof: Let C_k and C_{k+1} be two adjacent contours. Suppose, C_k and C_{k+1} did touch each other at some portion, then there will be some match (i, j) which belongs to both these contours. This implies that the length of the longest common subsequence ending at (i, j) is both k and $(k - 1)$. But this is a violation of the definition of contours, which says that the length of the longest common subsequence ending at a match $(i, j) \in C_k$ is k . Hence no two contours touch each other.

The other case we need to prove is that two contours never cross each other. If they did cross each other at some point, then there exists some $(i', j') \in C_{k+1}$, for which there is no match $(i, j) \in C_k$, which satisfies $(i < i')$ and $(j < j')$. This again violates the definition of contours since (i', j') no longer belongs to C_{k+1} as it cannot be chained with a sequence of k matches to obtain a *LCS* of size $(k + 1)$. ■

Definition 2.3 *Dominant matches* in a contour C_k are those matches $(i, j) \in C_k$ for which there is no other match $(i', j') \in C_k$ with $(i' = i \wedge j' < j)$ or $(i' < i \wedge j' = j)$ (indicated by the shaded circles in Fig. 2.1).

Lemma 2.2 *Let A and B be two sequences of length m and n with the matches represented in $m \times n$ matrix M . Let $C_1, C_2 \dots C_p$ be the contours obtained from M . Then there exists a longest common subsequence of length p comprising of only dominant matches*

Proof: This can be proved by showing that the longest common subsequence can be composed by going through contours in reverse order, i.e. from $C_p \dots C_2, C_1$. If $p = 1$ then we just have to pick one dominant match from the single contour. Now assume that we have constructed the longest common subsequence comprising of dominant matches from all the contours from C_p to C_2 . Let (i, j) be the latest dominant match (from C_2) included to this sequence. All the matches $(i', j') \in C_1$ satisfy either $(i' \leq i \wedge j' < j)$ or $(i' < i \wedge j' \leq j)$ and there is at least one match which satisfies $(i' < i \wedge j' < j)$. If not, C_1 touches or crosses C_2 , which violates the definition of contours. This match (i', j') with $(i' < i \wedge j' < j)$ can either be a dominant match or there exists a dominant match (i'', j'') such that $(i'' = i' \wedge j'' < j')$ or $(i'' < i' \wedge j'' = j')$. ■

Several algorithms have been suggested to find the longest common subsequence which make use of this contour based technique. The foremost among them is Hirschberg [4]. The other important work in this direction is that of Rick [14] and [13]. Bergroth and Hakonen [1] describe some of these contour based methods in their survey paper.

Chapter 3

Longest Tandem Subsequence(*LTS*)

The Longest Tandem Subsequence (*LTS*) for a given sequence $S = s_1s_2...s_n$ is a tandem sequence T such that $T \subseteq S$ and the length of the sequence T is the maximum possible. The *LTS* problem was first proposed by Kosowski [8]. In this chapter we briefly describe some previous and related works. We also provide an overview of our approach.

3.1 Previous Work

A related problem to *Longest Tandem Subsequence* is that of finding the longest *repetition* (substring) in a given input string. This problem (Longest Repetition) was first solved by Main and Lorentz [9], who gave an $O(n \log n)$ time algorithm. Kolpakov and Kucherov [7] describe with a linear time algorithm for finding all the maximal repetitions. Another relating problem is concerned with *Pattern discovery* or *Motif discovery* in DNA and other protein sequences. One recent work in this direction is that of Chattaraj [3].

The *Longest Tandem Subsequence* problem was put forth by Kosowski [8]. The algorithm given by him works in $O(n^2)$ time using $O(n)$ space. In his work, *LTS* problem for a sequence $S = s_1, s_2...s_n$ is solved in two stages, which are:-

- **STAGE 1:** Determine an index $l, 1 \leq l < n$ for which $|LCS([s_1...s_l], [s_{l+1}...s_n])|$ takes the maximum possible value.
- **STAGE 2:** Compute $T = LCS([s_1...s_l], [s_{l+1}...s_n])$ and return T as output.

The Stage 1 is achieved by using a *LTSsplit* algorithm. This algorithm forms the core of Kosowski's [8] work. The Stage 2 is solved using Hirschberg's [4] algorithm for longest

common subsequence.

3.1.1 Algorithm: *LTSplit*

The *LTSplit*¹ returns the index l for which the longest tandem subsequence is obtained. It is based on dynamic programming and works in $O(n^2)$ time. In this section we give a high level idea of how this algorithm works.

A family of functions f_k , for $1 \leq k \leq n$ is defined. For a given k function $f_k : \mathbb{Z} \times \mathbb{Z} \longrightarrow \mathbb{N}$ is defined as follows:

$$f_k(i, j) = \begin{cases} |LCS([s_1 \dots s_i], [s_j \dots s_k])| & \text{for } 1 \leq i < j \leq k \\ 0 & \text{in all other cases when } i, j \geq 0 \\ -1 & \text{when } i, j < 0 \end{cases}$$

In terms of f_k , index l is expressed as $f_n(l, l+1) = \max(f_n(r, r+1), 1 \leq r < n)$.

The values of function $f_k(i, j)$ for $1 \leq i < j \leq k \leq n$, is expressed using the following recursive formula:

$$f_k(i, j) = \begin{cases} \max\{f_k(i-1, j), f_{k-1}(i, j)\} & \text{when } s_i \neq s_k \\ f_{k-1}(i, j) + 1 & \text{when } s_i = s_k \end{cases}$$

One more function $d_k : \mathbb{N}^+ \times \mathbb{N} \longrightarrow \{0, 1\}$, for $(1 \leq k \leq n)$, is expressed as: $d_k(i, j) = f_k(i, j) - f_k(i-1, j)$ where, $(1 \leq i < j \leq k)$. Now, f_k can also be expressed as $f_k(i, j) = \sum_{r=1}^i d_k(r, j)$

Lemma 3.1 *The value of $d_k(i, j)$ is equal to 1 iff $j \in \langle i+1, a_k(i) \rangle$, for some function $a_k : \mathbb{N} \longrightarrow \mathbb{N}$*

Proof: Refer Kosowski [8] for proof. ■

The set of values for $A_k = \{a_k(i), \forall (1 \leq i \leq n)\}$ is found recursively from A_{k-1} (this is a consequence of the proof of Lemma 3.1). This process takes $O(n^2)$ time.

¹This part is taken verbatim from Kosowski[8]

The *LTSsplit* can now be summarized by the following two steps:-

1. For all $k, 1 \leq k \leq n$, compute the set A_k by modifying the set of values in A_{k-1} .
2. Determine the index l from the values in A_n using the following equation:

$$f_n(i, i+1) = \{p : (1 \leq p \leq i) \wedge a_n(p) \geq i+1\}$$

3.2 Our Approach

The Longest Tandem Subsequence (*LTS*) for a given sequence $S = s_1 s_2 \dots s_n$ is a tandem sequence T such that $T \subseteq S$ and the length of the sequence T is the maximum possible. The algorithm proposed in this thesis to find the *LTS* for a sequence $S = s_1, s_2 \dots s_n$ can be summarized as follows:

Step 1: Compute $T_{n/2} = LCS(S_L[s_1 \dots s_{n/2}], S_R[s_{n/2+1} \dots s_n])$. Let $p = |T_{n/2}|$.

Step 2: For($1 \leq l < p$) do

- 1: Shift one symbol from S_R to S_L (as well as from S_L to S_R).
- 2: Compute T_l , the *LCS* between $S_L[s_1 \dots s_{(n/2+l)}]$ and $S_R[s_{(n/2+l+1)} \dots s_n]$ (also the *LCS* between $S_L[s_1 \dots s_{(n/2-l)}]$ and $S_R[s_{(n/2-l+1)} \dots s_n]$)
- 3: If the current *LCS*, T_l , is the longest *LCS* obtained so far then $p \leftarrow |T_l|$ and $T \leftarrow T_l$.

Step 3: Return T as the *LTS*.

The contour based technique would serve as an efficient way to determine this index l , at which we get the *LTS*. This is because, the length of longest common subsequence between $S_L = s_1 s_2 \dots s_l$ and $S_R = s_{l+1} \dots s_n$ for an index $l, (1 \leq l < n)$ is indicated by the number of contours obtained from the matches between S_L and S_R . If $C_1, C_2 \dots C_{p_l}$ are the contours obtained, then p_l is the length of the longest common subsequence between S_L and S_R .

The algorithm discussed in the thesis makes use of this contour based technique. Initially, for $l = n/2$, the set of contours $C_1, C_2 \dots C_{p_{n/2}}$ for the two sequences $S_L = s_1 s_2 \dots s_{n/2}$ and $S_R = s_{(n/2+1)} \dots s_n$ is found. In the subsequent iterations we shift l rightwards (also leftwards) one symbol at a time for $(n/2 < l < (n - p_{max}))$ (as well as $(n/2 > l > p_{max}))$ times and re-adjust the contours at each step. Here, p_{max} indicates the length of the longest

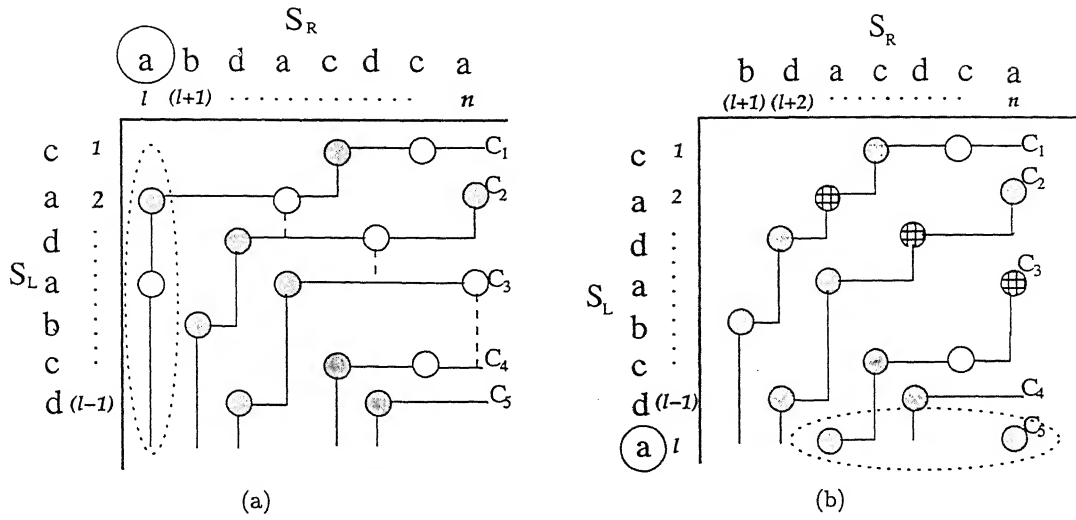


Figure 3.1: Contour Modification (after shifting a from S_R to S_L)

common subsequence obtained so far (which is nothing but the maximum number of contours obtained so far). The Fig. 3.1 shows the contour modification that takes place in any given iteration.

Chapter 4

The Algorithm

The algorithm comprises of two parts *Re-adjust Contours* and *Extend Contours*. *Re-adjust Contours* reconstructs the contours after each shift of l . After each shift, a symbol is added at the end of S_L (or at the beginning of S_R). Therefore, a new row (column) of matches is included. Accordingly, the contours have to be extended to include these matches (shown by the dotted ellipse in Fig. 3.1(b)). The algorithm *Extend Contours* serves this purpose. The sections 4.2 and 4.3 explain these steps in greater detail.(Refer Appendix A for the complete pseudocode)

4.1 Preprocessing

A list $Next_a[0..n-1]$ is constructed for each symbol $a \in \Sigma$, which gives for any i , ($0 \leq i < n$), $Next_a[i] = j$ such that $s_j = a$ with ($i < j \leq n$) and $\forall k, (i < k < j) s_k \neq a$. This is used to find the next earliest occurring instance of the symbol a after the index i . Computing such a list for each symbol takes $O(n)$ time. So for σ symbols, it takes $O(n\sigma)$ time.

Each match is denoted by an ordered pair (i, j) where $s_i = s_j$, ($1 \leq i < j \leq n$) and $(s_i \in S_L) \wedge (s_j \in S_R)$. Each contour is represented only by its dominant matches. All the dominant matches belonging to a contour are organised as a splay tree ([16],[10], [11]) with their respective second component (j) used as the key. The use of splay tree as the data structure has its advantages which is seen later in Sec. 4.2. The set of operations that can be performed on a splay tree are discussed in the Appendix B. In addition to the key, the

first component ' i ' of these matches is also stored in their respective nodes in the splay tree. The contours $C_1, C_2 \dots C_p$ are represented by their respective trees $T_1, T_2 \dots T_p$.

Lemma 4.1 *Given any two dominant matches $(i, j), (i', j') \in C_k$ such that $(j < j')$, we have $(i > i')$*

Proof: This lemma can be proved by contradiction. Consider the dominant matches (i, j) and (i', j') . We have the following two cases:

1. If $(i < i')$ then $(i, j) \in C'_k$ where $k' > k$.
2. If $(i = i')$ then (i, j) is not a dominant match.

So $(i > i')$. In general this inequality (or ordering) holds for any number of dominant matches belonging to the same contour. ■

As a consequence of the above lemma, we can say that the inorder traversal of any tree T_k will list the the values of key j in increasing order and the values of i in decreasing order.

The set of contours $C_1, C_2 \dots C_{p_{n/2}}$ for the two sequences $S_L = s_1 s_2 \dots s_{n/2}$ and $S_R = s_{n/2+1} \dots s_n$ is constructed by making use of *Extend Contours* (Sec. 4.3). The sequence S_L is extended one symbol at a time, from $i = 1$ through to $i = n/2$. In each iteration, the latest dominant match(it it exists) is always added to the leftmost position of its corresponding contour (this is by virtue of Lemma 4.1). So it would be feasible if we first construct the contours as linked list (Binary Search Tree growing in one direction). Later, when all the contours have been completely constructed, we build the corresponding balanced binary search trees. The time involved for these operations are later analysed in section 5.2.

4.2 *Re-adjust Contours*

Re-adjust Contours reconstructs the contours after shifting the leftmost symbol from S_R to the rightmost position in S_L . Let l be the current leftmost index of S_R . The column of matches corresponding to the shifted symbol (s_l) are deleted from C_1 to obtain C'_1 . This is accomplished by deleting the leftmost node in the tree T_1 . This deleted node, (i, l) ,

represents the dominant match of the column l . Let (i', j') be the next dominant match along the contour in the increasing order of the second component, that is $(l < j')$. This can be found by searching T_1 for j , by which the successor of j (which is j') is made the root of the tree (property of splay tree operations).

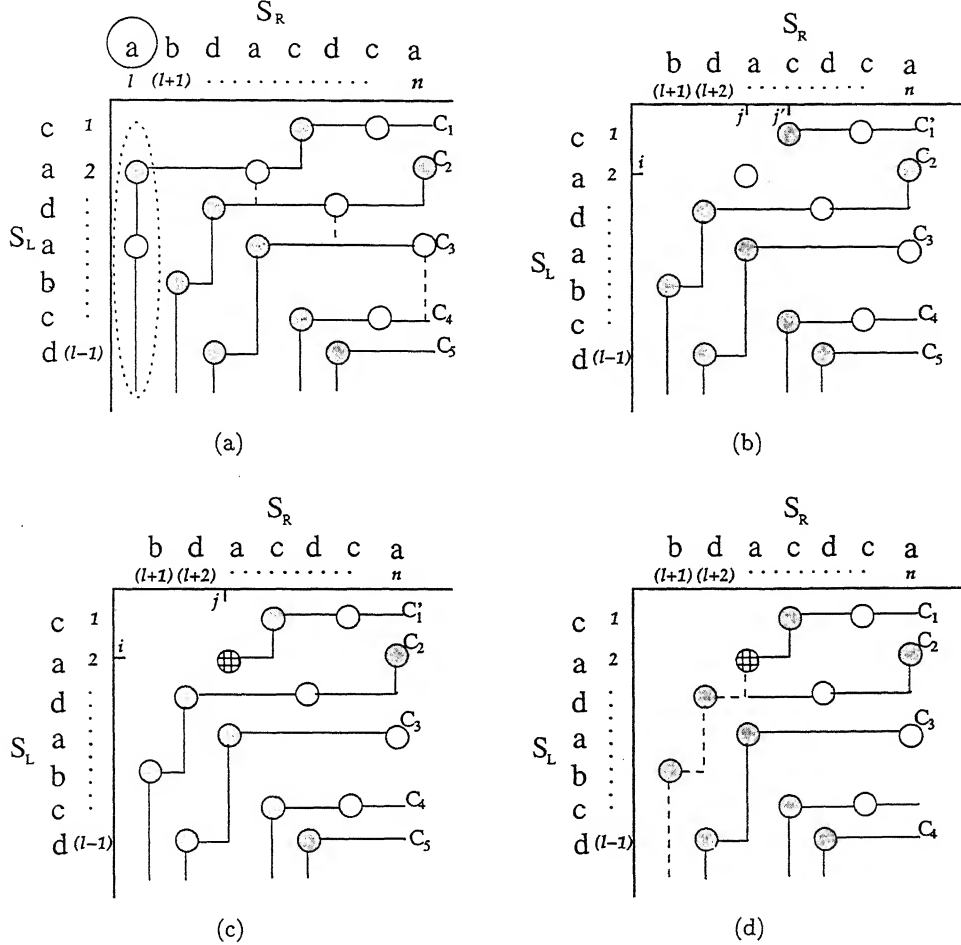


Figure 4.1: *Re-adjust Contours* (after removing a from S_R)

The contour C'_1 may not be a complete contour, as shown in Fig 4.1(b). So we check if C'_1 has to be updated to include a new dominant match. We first note that, if there is such a candidate match it has to be in that part of the contour C_1 between the two dominant matches (i, l) and (i', j') . Also, it cannot be along the column j' since (i', j') is already the

dominant match of this column. So the dominant match (if any) has to lie on the segment of the contour C_1 inbetween l and j' along the row i . The query $j = \text{Next}_{s_i}[l]$ returns the index of earliest occurrence of the symbol s_i after the index l . This match (i, j) , is nothing but the immediate next match after (i, l) along the row i . This candidate match is inserted into the tree T_1 as a dominant match if $(j < j')$. If not, it is ignored. The match in checkered pattern in Fig. 4.1(c) denotes such a kind of match which is included in T_1 as a dominant match.

By definition, C_k includes all those matches for which the length of the longest common subsequence ending at each of these matches is k . So every match in C_2 can be chained with at least one match in C_1 to get a common subsequence of length two. The matches in C_1 along l are lost when s_l is shifted from S_R to S_L . So all those matches in C_2 which could be chained only with the matches along l , no longer belong to C_2 . These matches are now included in C'_1 to get the complete contour C_1 .

Let (i, j) be the leftmost dominant match of C'_1 (match in checkered pattern in Fig. 4.1(c)). By Lemma 2.2, we have for every dominant match $(i', j') \in C_2$ with $(j' > j)$, some dominant match $(i'', j'') \in C'_1$ such that $(i'' < i') \wedge (j'' < j')$. So all such matches with $(j' > j)$ still belong to C_2 . The complete contour C_1 is obtained by joining that part of contour C_2 which extends from the left-end till j with C'_1 (the dashed polyline in Fig. 4.1(d)). This is done by splaying the tree T_2 at j , by which the left subtree contains all the dominant matches which are to the left of j and the right subtree contains all the matches to the right of j . This left subtree is joined (by making use of join operation of splay trees) with T_1 to obtain the complete contour C_1 .

The above set operations are repeated for the subsequent contours.

4.3 Extend Contours

After the contours are re-adjusted, they may have to be extended to include the row (or column) of matches. Let s_l be the symbol that is shifted from S_R to S_L , $n/2 < l < n$. By this, a new row l is included at the bottom of the matrix M (shown by the dotted circle in Fig. 3.1(b)). The contours $C_1, C_2 \dots C_{p_l}$ are processed from $(1 \leq k \leq p_l)$, one at a time. Let (i', j') be the leftmost dominant match of the the current contour C_k . Let (l, j) be the current leftmost match along the row l . We have three cases:-

1. If $(j' > j)$, then (l, j) is a dominant match in C_k , proceed to the next contour and also to the next match along the row l , immediately after the column j' .
2. If $(j' = j)$, then (l, j) is just match in C_k , proceed to the next contour and also to the next match along the row l .
3. If $(j' < j)$, then (l, j) is a part of some other contour, proceed to the next contour.

The remaining matches(if any) along the row l make up a new contour $C_{p(l+1)}$. The leftmost among these matches will be the dominant match for this contour. We maintain a pointer to the leftmost node in each tree $T_1, T_2 \dots T_{p_l}$. The next immediate match along the row l after the index j can be found using $Next_{s_i}[j]$ in constant time.

4.4 Shifting in the Reverse Direction

The algorithm explained so far shifts the index l rightwards one step at a time. A similar procedure can be used to shift the index leftwards (in the reverse direction). This procedure is of the same time complexity as that of the above algorithm. This procedure can be started off from the same set of initial trees $T_1, T_2 \dots T_{p_{n/2}}$ corresponding to the contours $C_1, C_2 \dots C_{p_{n/2}}$ (from Lemma 4.1).

When the rightmost symbol in S_L is shifted to the leftmost position in S_R , the row of matches corresponding to this symbol (s_l) is removed. This is done by deleting the leftmost dominant match, of the form (l, j) , in each contour, for $(l < j \leq n)$ (indicated by the dotted circle in Fig. 3.1(b)). The shift may also add a column of matches towards the left side of the matrix M (indicated by the dotted circle in Fig. 3.1(a)). The contours C_1 (through C_{p_l}) have to be restructured so that these matches (and subsequent matches) are included. Let (i, l) be the topmost match along the column l , we join this column with that part of contour C_1 that extends from i till the rightmost end. This is done by splaying T_1 at i and joining the right subtree of T_1 and the dominant match of this column (which is (i, l)) as the left subtree. If we already have a dominant match $(i, j) \in T_1$ where $(j > l)$, we just update this node to (i, l) . In the subsequent modifications of contours (from C_2 to C_{p_l}), the left subtree can comprise of more than one dominant match. After all the contours are modified, we may obtain one additional contour $C_{p(l-1)}$, where $p(l-1) = p_l + 1$

4.5 Computing the current *LCS*

The current longest common subsequence (a potential *LTS*) is computed whenever a new p_{max} is obtained. The *LCS* is obtained in the reverse order starting from $C_{p_{max}}$ and ending at C_1 . We take any dominant match in the inner most contour $C_{p_{max}}$. Let this match be (i, j) . By Lemma 2.2 we always have some dominant match $(i', j') \in C_{(p_{max}-1)}$ such that $((i' < i) \wedge (j' < j))$. This match can be found by searching for j in the tree $T_{(p_{max}-1)}$. This search returns the predecessor of j in contour $C_{(p_{max}-1)}$ which is the match (i', j') we are looking for. This search is repeated for the subsequent contours too, to obtain the chain of dominant matches in the reverse order. These dominant matches when rearranged in proper order gives us the current longest common subsequence. The time required for this process is analysed later in section 5.2.

Chapter 5

Correctness and Analysis

5.1 Correctness of the Algorithm

In order to prove the correctness of the algorithm, the invariant that has to be observed here is that, none of the contours touch or cross each other. The set of contours $C_1, C_2 \dots C_p$ can be seen as a set of ordered lists. After each shift, the first element of C_1 is deleted to obtain a partial contour C'_1 . By definition, C_k includes all those matches which correspond to the k^{th} match from the start of the longest common subsequence (till that match). So C_1 represents the set of all first matches. Since, at each shift, the matches along the leftmost column are removed, C_1 is updated so that it includes all the first matches. The algorithm is achieving this by splaying C_2 with respect to the first element of C'_1 . Also, the algorithm does not stop just at this, it repeats the procedure for the subsequent contours.

The invariant that no two contours touch or cross each other is also maintained. This is evident because the left subtree after the splay operation in k^{th} iteration is joined with exactly one other tree, corresponding to the partial contour C'_{k-1} . So the dominant matches corresponding to the left subtree belong to exactly one contour C_k . Since the contours are processed in order ($1 \leq k \leq p_l$), there will be no crossings.

5.2 Analysis

The preprocessing involves computing the list $Next_a[0 \dots n-1]$ for each symbol $a \in \Sigma$. This takes $O(n\sigma)$ time, where σ is the size of the alphabet Σ . If σ is $O(n)$, we make use of the data structure described in Chapter 6. Since we are storing only the dominant matches

in the trees $T_1, T_2 \dots T_p$, we consider only the number of dominant nodes while analyzing the complexity involved in the tree operations. Rick [13] gives the following bound on the number of dominant matches each contour can have. For two sequences $A = a_1 a_2 \dots a_m$ and $B = b_1 b_2 \dots b_n$, ($m \leq n$), if p is the length of the longest common subsequence, then we cannot have more than $((n - p) + (m - p) + 1)$ dominant matches per contour. In case of *LTS* there is a single string $S = s_1 s_2 \dots s_n$ of length n . Let the length of the longest tandem subsequence be p_{max} . So the bound on the number of dominant matches per contour in this case is $(n - 2p_{max} + 1)$.

Each splay tree operations take $O(\log n)$ amortized time. The time required for constructing the trees for the initial set of contours $C_1, C_2 \dots C_{p_{n/2}}$ for the two sequences $S_L = s_1 s_2 \dots s_{n/2}$ and $S_R = s_{n/2+1} \dots s_n$ is

$$O(np_{n/2} + p_{n/2} d_{n/2} \log d_{n/2}),$$

where, $d_{n/2} = (n - 2p_{n/2} + 1)$ is the maximum number of dominant matches each of these contours can have.

At each iteration we add one symbol to S_L using *Extend Contours*. This may involve adding a dominant match to at most of $p_{n/2}$ lists. This process is done $n/2$ times. Since the contours are always growing in the leftward direction, the dominant match is always inserted at that beginning of the corresponding lists. This takes constant time. So totally, to construct the complete set of contours we require $O(np_{n/2})$ time. Once these lists are constructed we require $O(d_{n/2} \log d_{n/2})$ to convert each of them to a balanced Binary Search Tree(BST). Since there are $p_{n/2}$ such lists, totally we require $O(p_{n/2} d_{n/2} \log d_{n/2})$ to construct all the balanced BSTs.

Let $p = p_{max}$ be the length of the longest tandem subsequence, ($0 \leq p \leq n/2$). Let $d = (n - 2p + 1)$ represent the bound on the dominant matches. Each pass through *Re-adjust Contours* takes $O(p \log d)$ time, since we may re-adjust at most p contours and each of these re-adjustments take constant number of splay operations. The *Re-adjust Contour* operation is performed each time a symbol is shifted from S_R to S_L or vice versa. We shift index l till p^{th} position on the left and till $(n - p)^{th}$ on the right. So, totally there are $(n - 2p) = O(d)$ shifts. So total number of operations are at most

$$(d \times p \times \log d)$$

The time complexity of for each iteration of *Extend Contour* is $O(p \log d)$. So the overall complexity of this step is $O(dp \log d)$.

The time take by the $LCS(S_L[s_1 \dots s_l], S_R[s_{l+1} \dots s_n])$, which gives the longest tandem subsequence obtained so far, is $O(p \log d)$.

The time complexity of our algorithm to find the Longest Tandem Subsequence is

$$\begin{aligned} &O(n\sigma + np_{n/2} + p_{n/2}d_{n/2} \log d_{n/2} + dp \log d + p \log d) \\ &= O(n\sigma + dp \log d). \end{aligned}$$

Space Complexity: The data structure $Next_a[1 \dots n]$, $(\forall a \in \Sigma)$ takes $O(n\sigma)$ space. We have p trees each of which contain at most $d = (n - 2p + 1)$ nodes. So the space required for this is $O(dp)$. Hence, the space complexity of the algorithm is

$$O(n\sigma + dp).$$

Chapter 6

A New Data Structure for the *LCS* Problem

6.1 Introduction

Given two sequences $A = a_1a_2...a_m$ and $B = b_1b_2...b_n$, $m \leq n$, over some alphabet Σ of size σ , the Longest Common Subsequence (*LCS*) problem is to find a subsequence of greatest possible length that occurs in both A and B . In this chapter, we discuss a problem which is a special case of the *LCS* problem. Consider a scenario where we have a long *Document sequence* $D = d_1d_2...d_n$, defined over a large alphabet Σ of size σ . The document sequence D is queried (*LCS* search) at different instances by different short *Query sequence*, $Q = q_1q_2...q_m$.

An important application is finding a consensus among DNA sequences[15]. The genes related to proteins synthesis evolve with time, but the functional regions remain consistent in order to work correctly. In order to find these functional regions, which remain unchanged through time, we make use of *LCS* algorithms. Another important related application is when biologists discover new genes, they wish to ascertain to which other known genes these newly discovered genes are similar. This is vital for identifying the function of a newly discovered gene.

In this chapter, we design a space efficient data structure to represent the document sequence. The data structure also supports time efficient processing of the query. In

general the data structure described can be used to find the *LCS* between any given sequences A and B , as long as the given alphabet Σ is large i.e. the size σ is of the order $\Omega(n/\log^2 n)$.

6.2 Related Work

The Longest Common Subsequence *LCS* has been extensively studied for over three decades. A number of algorithms were suggested over the years. The performance of each of these algorithms depended on different parameters like the length of the *LCS* (or the edit distance between the two input sequences), the size of the alphabet, the number of matches between the two sequences etc. The Table 6.1 gives a summary of space and time complexities of some of the well-known algorithms. In the table, m and n are the lengths of the two input sequences, A and B , where $m \leq n$; p denotes the length of the *LCS* between A and B ; d denotes the bound on the maximum number of dominant matches in each contour, ($d \leq (n - p) + (m - p) + 1$), this is from [14].

Work	Time Complexity		Space Complexity	
	Preprocessing	Overall	Data Structure	Runtime
Hirschberg [5]	$O(n \log \sigma)$	$O(np + n \log n)$	$O(n)$	*
Nakatsu et al. [12]	-	$O(m(n - r))$	-	$O(mn)$
Hsu, Du [6]	$O(n \log \sigma)$	$O(pm \log(n/m) + pm)$	$O(n)$	$O(pd)$
Rick [13]	$O(n\sigma)$	$O(n\sigma + \min\{mp + p(n - p)\})$	$O(n\sigma)$	$O(n\sigma)$

Table 6.1: Previous work on *LCS*

The first efficient algorithm was suggested by Hirschberg [5]. Though the algorithm has a good time complexity of $O(np + n \log n)$, test results from Bergroth et al. [1] show that the runtime memory requirement for this algorithm is very high. As pointed out by Rick [14] the algorithm also has a high constant factor associated with it's time complexity. The algorithm by Nakatsu et al. [12] involves little preprocessing, however, it has a space requirement of the order of $O(mn)$. Bergroth et al. [2] have tried to refine this algorithm through some heuristics. These heuristics try to estimate a lower bound and an upper bound on the length of the output. Though a little efficiency in terms of space is achieved, the time complexity nevertheless is more. The time complexity of Hsu, Du [6] is comparable to that of ours, but just like the algorithm in [5], the constant factors involved are high.

The performance of Rick's [13] is very good, but it relies on a precomputed table of size $O(\sigma n)$ which requires $O(\sigma n)$ space. This overhead becomes high if the size of the alphabet, σ is large. Our algorithm performs better when $\sigma \geq (n/\log^2 n)$ and $(m \ll n)$ (see Section 6.7 and Appendix C for a comparative analysis).

6.3 Preliminaries

We define two sequences, a *Document sequence* $D = d_1 d_2 \dots d_n$ and a *Query sequence*, $Q = q_1 q_2 \dots q_m$, where $n \gg m$, over an alphabet Σ of size σ . The concepts of *matches*, *contours*, *dominant matches* are similar to those used in Chapter 2.

Definition 6.1 A *Predecessor* of a dominant match $(i, j) \in C_k$ is a dominant match $(i', j') \in C_{k-1}$, such that $(i' < i \wedge j' < j)$.

Definition 6.2 The *Closest Predecessor* of a dominant match $(i, j) \in C_k$ is a dominant match $(i', j') \in C_{k-1}$, such that $(i' < i \wedge j' < j)$, and if there exists some dominant match $(i'', j'') \in C_{k-1}$, such that $(i'' < i \wedge j'' < j)$, then $(i'' < i' \wedge j'' > j')$.

6.4 The Document Search Tree (*DST*)

The only preprocessing that is done is construction of the *Document Search Tree*. The *document sequence* $D = d_1 d_2 \dots d_n$ is organized as a search tree. This tree is used for finding the immediately next occurring instance of a symbol 'a' in the sequence D , after a given index j . This is required for detecting the dominant matches (as discussed in Section 6.5.2). In this section we are going to describe the construction and operation of this *Document Search Tree (DST)*. A search on the *DST* takes $O(\log \log n)$ time.

6.4.1 Construction

■ Leaf Nodes

The sequence $D = d_1 d_2 \dots d_n$ is divided into $\log n$ *segments* of $(n/\log n)$ symbols each. Let these segments be called $X_1, X_2 \dots X_{\log n}$. The segments $X_1, X_2 \dots X_{\log n}$ are stored as leaves of the *DST*.

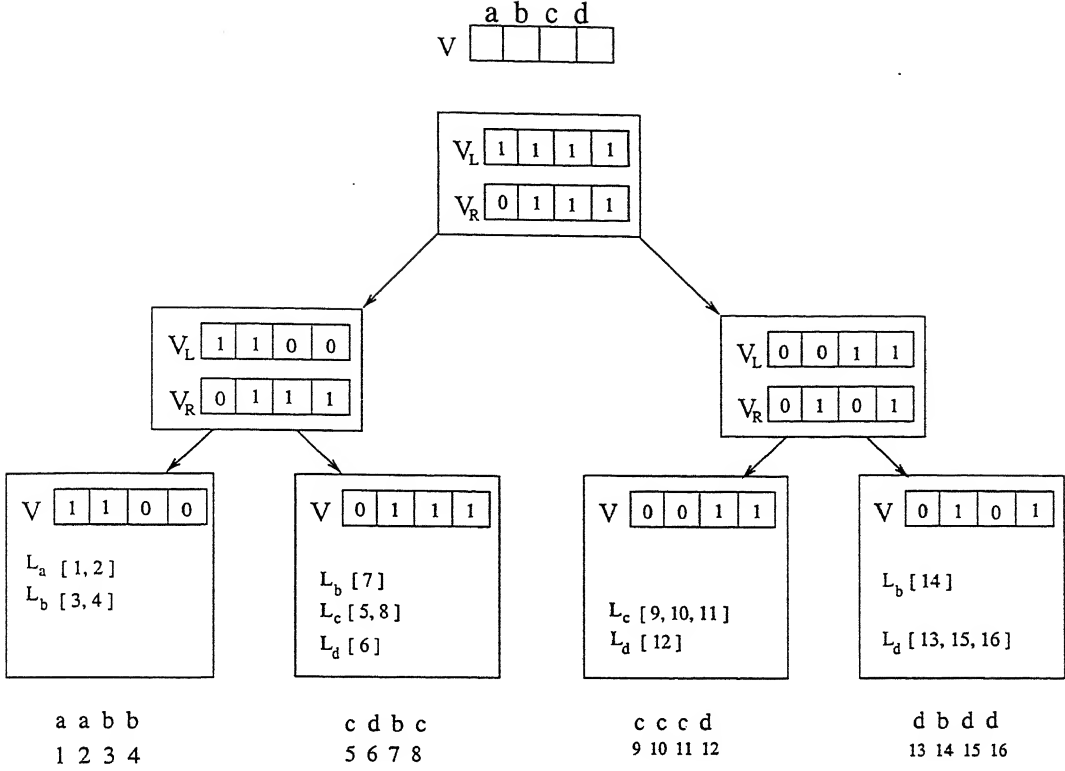


Figure 6.1: *DST* for the sequence $D = aabbcdccccdbdd$

Each leaf X_l , ($1 \leq l \leq \log n$) is organized as a set of (at most) $\sigma = |\Sigma|$ lists. These lists can be of two kinds (as shown in Fig. 6.2), based on the frequency f_a (number of occurrence) of each alphabet $a \in \Sigma$ in X_l . They are:

Case 1: If ($0 < f_a \leq \log n$), then store the indices (in increasing order) of occurrences of a as a simple array L_a . Let this list be called *Type-I* list. We can locate the nearest occurrence of any symbol a in $O(\log f_a) = O(\log \log n)$ time (see Section 6.4.2 for details).

Case 2: If ($f_a > \log n$), then a list $L_a[0 \dots ((n/\log n) - 1)]$ is constructed in which at each index j of L_a we store the index of next occurring instance of a in X_l immediately after j . Let this be called *Type-II* list. A search of the nearest occurrence on this list takes $O(1)$ time (see Section 6.4.2 for details).

In addition to these lists a bit vector V of size σ (as shown in Fig. 6.1) is stored in each of these leaves. In a leaf X_l , the bit corresponding to a symbol a in the bit vector V is set

to 1 if L_a is not empty, otherwise it is set to 0. The organization of the DST is shown in Fig. 6.1

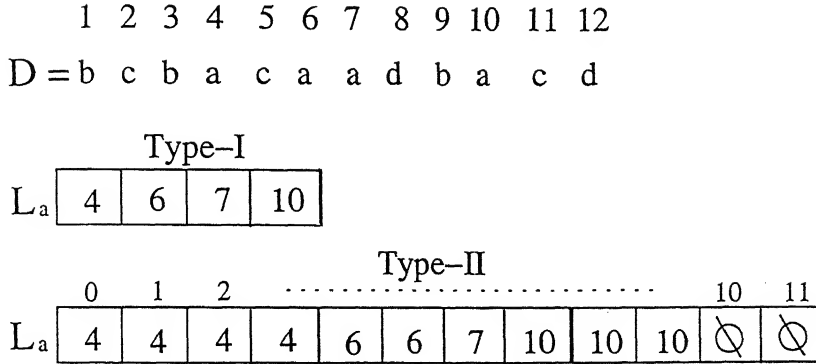


Figure 6.2: Type-I and Type-II Lists for symbol a

■ Non-leaf nodes

Each internal (nonleaf node) contains two bit vectors V_L and V_R . The bit vector V_L of an internal node is obtained through logical OR operation of the two bit vectors stored in its left child. Similarly, the bit vector V_R is obtained through logical OR operation of the two bit vectors stored in its right child. If the left child (or right child both) of an internal node is a leaf node the vector V_L of this internal node is assigned the same value as the vector V of its left child (or right child or the respective children). The Fig. 6.1 shows the bit vectors of all the nodes in the DST .

6.4.2 Searching the DST

Given an index j and a symbol $a \in \Sigma$, the DST can be used to efficiently search the index j'' of the next occurrence of a in $D[d_1d_2 \dots d_n]$ immediately after j . We first locate into which of the segments j falls under. This segment l is obtained from $\lceil j / \log n \rceil$. In the segment X_l , the list L_a can be either of Type-I or Type-II as discussed in Section 6.4.1. Accordingly:

Case 1 (Type I List): We check the list L_a to see if there is an occurrence of a after j . This takes constant time, since we only have to check the last entry in L_a . If this entry has a value greater than j , then surely L_a contains j'' (otherwise j'' might

be located in some subsequent segment $l' > l$). The list L_a is searched for j to locate j'' . This would take at most $O(\log \log n)$ time, since the list is of size at most $\log n$.

Case 2 (Type II List): The index $j(1 \leq j \leq n)$ is first translated to $j = j - (l * (n / \log n))$. If $(L_a[j] \neq \phi)$, then the required index is $(j'' = L_a[j])$ (otherwise j'' may be located in a subsequent segment $l' > l$). This index j'' is translated back to $(j'' = j'' + (l * (n / \log n)))$. The overall time taken in this case is $O(1)$.

The other case is, j'' is located in some other segment. To locate this segment we start from the leaf X_l keep on moving up the DST till we encounter the first internal node, in which the bit vector V_R has $V_R[a] = 1$ (the bit corresponding to the symbol a set to 1). Starting from this node we move downwards. Since we are searching for the next immediate occurrence of symbol a , we try keep towards left as much as possible, i.e. we check at each node (during the downward traversal) if $V_L[a] = 1$, if so, we proceed towards left, otherwise we take the right child and continue to keep towards left as much as possible. Finally, when the leftmost leaf $X_{l'}$ (after j containing an instance of symbol a) is reached, we return the first entry in the list L_a (irrespective of whether the list is of Type-I or Type-II) inside the leaf $X_{l'}$. Since we ascend and descend the tree for at most $\log \log n$ levels, the time involved here is $O(\log \log n)$

6.5 The Algorithm

The algorithm discussed in this section involves very little preprocessing. The same contour-based approach (as discussed in Chapter 2) is employed for finding the *LCS* between $D = d_1 d_2 \dots d_n$ and $Q = q_1 q_2 \dots q_m$. The contours $C_1, C_2 \dots C_p$ are maintained as stacks $S_1, S_2 \dots S_p$. Here p denotes the length of the longest common subsequence between Q and D . The contents of the stack are compound nodes (not just a single element). Whenever a dominant match $(i, j) \in C_k$ is discovered the following two things are pushed onto the stack S_k :

- the match (i, j)
- the index to the closest predecessor of (i, j) in S_{k-1}

So the stacks contain only dominant matches and the index to their closest predecessors.

The *LCS* between Q and D can be easily traced once all the contours have been constructed completely. We just need to choose a dominant match in the inner most contour C_p from its corresponding stack S_p and trace back to its closest predecessor in S_{p-1} . The process is continued further till S_1 is reached. This would give us the *LCS* in the reverse order (see Lemma 2.2).

6.5.1 Constructing the Contours

The algorithm works as follows. The sequence $Q = q_1q_2\dots q_m$, which are aligned as rows in the matrix M is processed one symbol at a time. We maintain a list of indices t_1, t_2, \dots, t_p corresponding to the top of the stacks S_1, S_2, \dots, S_p respectively. This list is updated after each iteration i.e. after *all* the matches along the current row have been processed. The dominant matches in the current row are pushed onto their respective stacks, along with the indices to their closest predecessors. The closest predecessor of a dominant match $(i, j) \in S_k$ is nothing but the top most node t_{k-1} of the stack S_{k-1} . This can be shown as follows. Let the (i', j') be the match corresponding to t_{k-1} . We have $(i' < i)$ because t_{k-1} points to the top of the stack S_{k-1} (the most recent dominant match pushed) at the previous iteration $(i-1)$. So $(i' \leq i-1)$. We also have $(j' < j)$. Since, if $(j' = j)$ then the match $(i, j) \in C_{k-1}$, which is a contradiction. *After the current row is processed completely*, we update t_1, t_2, \dots, t_p to point to the current top of their respective stacks and move on to the next row (symbol).

6.5.2 Detecting the Dominant Matches

The process of detecting the dominant matches is similar to the one discussed in Section 4.3. The only difference is that, here we incur an additional search cost to detect the dominant match along the current row. Let i be the current row being processed. The contours C_1, C_2, \dots, C_p are processed from $(1 \leq k \leq p)$, one at a time. Let (i', j') be the leftmost dominant match of the current contour C_k . This is nothing but the top of stack S_k . Let (i, j) be the current leftmost match along the row i . We have three cases:

1. If $(j' > j)$, then (i, j) is a dominant match in C_k , proceed to the next contour and also to the next match along the row i , immediately after the column j' .
2. If $(j' = j)$, then (i, j) is just match in C_k , proceed to the next contour and also to the next match along the row i .

3. If $(j' < j)$, then (i, j) is a part of some other (inner) contour, proceed to the next contour.

A search is involved for the case 1, where we need to locate the next match along the row i , immediately after the column j' . Let q_i represent the symbol corresponding to row i . We need to search for the index j'' of next occurrence of the symbol q_i (immediately after j') in $D = d_1 d_2 \dots d_n$. This is done by making use of the *Document Search Tree* described in Section 6.4. Once we have got this new leftmost match (i, j'') , we again evaluate in which of the above three cases the current instance falls under, and the process is repeated for the subsequent contours. The remaining matches(if any) along the row i make up a new contour C_{p+1} . The leftmost among these matches will be the dominant match for this contour.

6.6 Analysis

6.6.1 Preprocessing

There are $\log n$ segments $X_1, X_2 \dots X_{\log n}$ each of size $(n/\log n)$ which are stored as leaves in DST . In each leaf X_l , $(1 \leq l \leq \log n)$, we have a list $L_a, (\forall a \in \Sigma)$, either of Type-I or Type-II. The Type-I list has size of at most $\log n$ while each Type-II list has exactly $n/\log n$ entries. Each of this segments has a bit vector V of size σ . We need $O(\sigma)$ time to construct each such bit vector. So to construct $\log n$ vectors we need $O(\sigma \log n)$ time.

Remark: Consider the case, when $\sigma \geq (n/\log^2 n)$. The size of each leaf X_l is $n/\log n$. If the symbols are distributed uniformly among the leaves, then in each leaf, each symbol has $((n/\log n)/\sigma) \leq \log n$ average occurrences. Therefore each $L_a, (\forall a \in \Sigma)$ can be represented as Type-I list. In Type-I lists we store each index exactly once. So at each leaf the total complexity of constructing σ lists is $O(n/\log n)$. Since we have $\log n$ leaves, the construction of all the lists takes $O(\log n \times (n/\log n)) = O(n)$. This is the best case space and time requirement for all the leaves in the data structure DST .

The other case is when in each leaf X_l , we store each $L_a, (\forall a \in \Sigma)$ as a Type-II list. This means that the number of occurrences of each symbol in every leaf is more than $\log n$. So we store all the σ lists as lists of size $n/\log n$ exactly. In this case we require a total of

$(\sigma \times n / \log n \times \log n)$, which is of the order $O(n\sigma)$. This is the worst case space and time requirement for all the leaves in the data structure *DST*.

In the *DST* we have $\log n$ leaves. So the depth of the tree would be $O(\log \log n)$. The total number of nodes in the tree are of the order $O(\log n)$. The construction of tree therefore takes $O(\sigma \log n)$ time, where the σ factor is for computing the bit vectors V_L and V_R at each node. The space required for storing all the bit vectors in the *DST* is of the order of $O(\sigma \log n)$.

To sum up everything, construction of the *DST* requires $O(n + \sigma \log n)$ space and time in the best case. This is when $\sigma \geq (n / \log^2 n)$. The complexity now becomes $O(n + n / \log n = O(n))$. Hence the best case Time and Space Complexity for constructing the *DST* is $O(n)$. The worst case time and space complexity for constructing the *DST* is when $\sigma < (n / \log^2 n)$. This is of the order $O(n\sigma)$.

6.6.2 The Algorithm

The construction of the contours takes $O(mp \log \log n)$ time. The entire query sequence $Q = q_1 q_2 \dots q_m$ is processed one symbol after the other. So we have m iterations. In each iteration we may add at most one dominant matches to each of the p contours. So all the p contours have to be checked. This checking may involve a search in the *DST* which requires $O(\log \log n)$ time.

Rick [13] shows that the maximum number of dominant matches, d , any contour can have is $d = (n - p) + (m - p) + 1$. Since we are storing only the dominant matches the space requirement is $O(pd)$. The time required to report an instance of the *LCS* is $O(p)$. Since we store the index to the closest predecessor match for each dominant match.

6.7 Comparison with the existing algorithms

The advantage of our algorithm can be seen when the size of the alphabet is large i.e. $\sigma \geq (n / \log^2 n)$. We assume that the alphabet is uniformly distributed through the input $D = d_1 d_2 \dots d_n$. The Table 6.2 gives a comparative analysis.

It is evident from the table and also from the results shown in Appendix [?] that the algorithm in [13] suffers from the preprocessing overhead, when $\sigma \geq (n / \log^2 n)$. The *Algorithm 2* in [13] and the algorithm presented in this chapter were implemented and

Work	Time Complexity		Space Complexity	
	Preprocessing	Overall	Data Structure	Runtime
Hirschberg [5]	$O(n \log n)$	$O(np + n \log n)$	$O(n)$	*
Nakatsu et al. [12]	-	$O(m(n - r))$	-	$O(mn)$
Hsu, Du [6]	$O(n \log n)$	$O(pm \log(n/m) + pm)$	$O(n)$	$O(pd)$
Rick [13]	$O(n^2 / \log^2 n)$	$O((n^2 / \log^2 n) + \min\{mp + p(n - p)\})$	$O(n^2 / \log^2 n)$	$O(n^2 / \log^2 n)$
Our Algorithm	$O(n)$	$O(n + mp \log \log n)$	$O(n)$	$O(pd)$

Table 6.2: Comparative Analysis of *LCS* algorithms($\sigma \geq (n / \log^2 n)$)

tested for different input size (m and n , ($m \ll n$)) and different alphabet size (σ). The input sequences were generated randomly. The algorithms were implemented in C++ (Red Hat - 9.0, GNU C compiler (gcc version 3.2.2)) on a Pentium 4(2.4 GHz), 256 MB RAM machine. The Appendix C gives a comparative study of the running times of both these algorithms.

Chapter 7

Different Search Methods

7.1 Introduction

In the previous chapter, we saw how the *DST* is used to perform a search in $O(\log \log n)$ time, to locate the next occurring instance of an alphabet a . In this chapter, we discuss some variations of the data structure, according to some specified *Search Cost* λ , with the search time $O(\lambda)$ between $O(1)$ and $O(\log n)$. We first describe how the search is performed in $O(1)$ and $O(\log n)$ time. We define the following terms before proceeding further:

$D[d_1d_2\dots d_n]$ – The input Sequence.

Σ – The input alphabet.

σ – The size of the input alphabet.

w – The number of machine words required to store σ bits.

λ – The search cost.

k – The number of permissible levels of *DST*.

f_a – The total number of occurrence of the symbol a in the input sequence D .

$f_a(X_l)$ – The frequency of occurrence of the symbol a in a the segment(or leaf) X_l .

7.2 Different Search Methods

In this section we discuss different searching methods. As in Section 6.4.1 we define the following kinds of lists.

Definition 7.1 The *Type-I* list of an alphabet a is a simple array which stores the indices of occurrences of a in the increasing order (see Fig. 7.1)

In order to locate the next occurring instance of a after a given index j , a binary search is performed on the Type-I list. This takes $O(\log n)$ time.

1 2 3 4 5 6 7 8 9 10 11 12
 $D = b \ c \ b \ a \ c \ a \ a \ d \ b \ a \ c \ d$

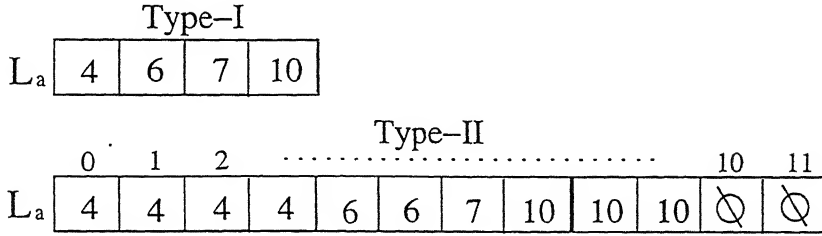


Figure 7.1: Type-I and Type-II Lists for symbol a

Definition 7.2 The *Type-II* list of an alphabet a is a simple array of size n where at each index i we store the index of immediate next occurring instance of a . (see Fig. 7.1)

In order to locate the next occurring instance of a after the given index j , we just have to access the j^{th} element in the Type-II list of the symbol a . This is a constant time operation.

7.2.1 Search Cost $\lambda = O(1)$

Construct Type-II lists ($\forall a \in \Sigma$). This takes $O(n\sigma)$ space and time.

7.2.2 Search Cost $\lambda = O(\log n)$

Construct Type-I lists ($\forall a \in \Sigma$). This takes $O(n)$ space and time.

7.2.3 Search Cost $k\lambda$, Levels of $DST = k$

The main idea is to construct the DST s recursively. In the previous description of the DST , as soon as we reached a depth of $O(\log \log n)$, we constructed the leaf nodes for each segment X_l .

The idea here is that, we continue the process of constructing the DST upto a maximum of k levels. At the end of each level, we check in each segment X_l , if the $f_a(X_l)$, ($\forall a \in \Sigma$), is less than $2^{k\lambda}$, if so we construct Type-I lists for each of these alphabets. For the remaining symbols in X_l we construct DST s recursively with the level reduced by one.

The input sequence $D[d_1d_2\dots d_n]$ is divided into 2^λ segments each of size $n/2^\lambda$. Let $f_a(X_l)$ be the frequency of occurrence of the symbol a in the segment X_l . We perform the following steps for each segment X_l ($1 \leq l \leq 2^\lambda$).

Step 1: ($\forall a \in \Sigma$) where $(f_a(X_l) \leq 2^{k\lambda})$, Construct Type-I lists for each such symbol a .

Step 2: ($\forall b \in \Sigma$) where $(f_b(X_l) > 2^{k\lambda})$.

If($Level \geq 1$) then, construct *DST* with *Search Cost* = $2^{(k-1)\lambda}$ and $Level = Level - 1$.

If($Level = 0$) then, construct Type-II lists for all such b 's.

Analysis: In the worst case the input sequence $D[d_1d_2\dots d_n]$ of size n may be divided k times into $(2^\lambda)^k = 2^{k\lambda}$ segments. In each of these $2^{k\lambda}$ segments ($\forall a \in \Sigma$), the frequency $f_a(X_l)$ may be greater than $2^{k\lambda}$. So we have a Type-II list for each of these symbols. So in the worst case we can have a data structure comprising of k levels of *DST* and all the lists stored as Type-II lists. The Type-II lists for all the symbols take $O(n\sigma)$ space. If k levels of *DSTs* are constructed in depth first manner, i.e. constructing all the lists in the left most leaf before proceeding to the next leaf (rightwards), then we require $O(kn + n\sigma)$ time to construct all the leaves. The depth of one *DST* is $O(\log 2^\lambda) = O(\lambda)$. Since we have k such *DSTs*, the depth of the entire structure is $O(k\lambda)$. In the complete data structure we have $O(2^{k\lambda})$ leaves. We store a bit vector for each of these leaves and in all other internal nodes too. So the total number of bit vectors is of the order $O(2^{k\lambda})$. So the total time required for obtaining all the bit vectors in the entire structure is $O(w(2^{k\lambda}))$. Hence, in the worst case the time complexity for constructing the *Extended-DST* is

$$O(kn + \sigma n + w2^{k\lambda})$$

and the space complexity is

$$O(\sigma n + w2^{k\lambda})$$

Example Consider the case, when $n = 2^{10}$, $\sigma = 3$, $k = 2$ and $\lambda = (\log \log \log n)$. Since $k = 2$ in the worst case, the input sequence of size n is divided twice into $\log \log n$ segments. So the total number segments are $(\log \log)^2 n$. Since we have $n = 2^{10}$, the total number of segments is 93. In each of these 93 segments we have around $2^{10}/93 \approx 11$ symbols per segment.

In each of the segments (of size 11) we can have $f_a(X_l) > 2^{k\lambda}$ ($\forall a \in \Sigma$). This is because, $11/3 > 2(\log \log 10)$. Thus, in the worst case we have Type-II lists for all the symbols. ■

7.2.4 Search Cost = λ , Unrestricted number of Levels

In the previous section, we constructed *DST* upto k levels. At each level we filtered off those symbols which have a frequency of occurrence (in that segment) less than $2^{k\lambda}$. The symbols remaining (in each segment) after constructing k levels of *DST* were represented as Type-II.

Now, consider the case in which the number of levels are unrestricted, that is, we keep on constructing *DSTs* recursively till in each segment the frequency of occurrence for each symbol in that segment is less than the *Search Cost*. At which point, we make use of Type-I list for storing the indices.

Analysis: In the worst case, we could have a tree in which all the leaves are located at the same depth from the root (say a depth of $k\lambda$). In each leaf (X_l) of this tree, we have ($\forall a \in \Sigma$), $f_a(X_l) = 2^\lambda$. So we have σ Type-I lists each of size 2^λ . Thus the total number of symbols in each leaf is $\sigma \times 2^\lambda$. At each level we divide a segment from the previous level into 2^λ segments. If n is the size of the input sequence and is of the form $n = ((2^\lambda)^k \times \sigma)$ for some k , then we have k levels of *DST* in which each leaf contains $\sigma 2^\lambda$ symbols. These symbols are represented as Type-I lists. The total space required for all the Type-I lists over all the leaves is $O(n)$. Since, there are k levels of *DST*, we have $2^{k\lambda}$ leaves and $O(2^{k\lambda})$ nodes. The space requirements for all the bit vectors is $O(w2^{k\lambda})$. The total space complexity is

$$O(n + w2^{k\lambda})$$

The time complexity to construct the *Extended-DST* with unrestricted number of levels is

$$O(kn + w2^{k\lambda})$$

7.2.5 Selective Representation

In this section we discuss another way of constructing the search data structure for each symbol a in Σ based on the frequency f_a . We have a separate structure for each ($a \in \Sigma$). Let the search cost be λ . We make use of Type-I list to represent each symbol a if ($f_a < 2^{k\lambda}$)

and Type-II lists for each symbol b with $(f_b \geq 2^{k\lambda})$. Let F_a be the total number of Type-I lists and F_b be the total number of Type-II lists. The size of the input sequence is n , we have:

$$\begin{aligned} n &= \sum_{(f_a < 2^{k\lambda})} f_a + \sum_{(f_b \geq 2^{k\lambda})} f_b \\ &\geq \sum_{(f_b \geq 2^{k\lambda})} f_b \\ &\geq (2^{k\lambda}) F_b \\ \therefore F_b &\leq n/(2^{k\lambda}) \end{aligned}$$

Since the total number of symbols in the alphabet is σ , we have

$$F_b \leq \min(\sigma, n/(2^{k\lambda}))$$

If $(k\lambda \leq \log(n/\sigma))$ then $F_b \leq (n/2^{k\lambda})$.

The space complexity of this search structure is $O(\min(n\sigma, n^2/(2^{k\lambda})))$, as we require $O(n)$ space for each Type-II list. The time required for this selective construction is also $O(\min(n\sigma, n^2/(2^{k\lambda})))$ (the time required for constructing all the Type-II lists).

7.3 Comparison of the Different Search Methods

The following table (7.1) summarizes the space and time complexities of the different search methods. The size of the input sequence $D[d_1d_2\dots d_n]$ is indicated by ' n ', ' σ ' indicates the size of the alphabet, ' f_{\max} ' represents the number of occurrences of the most frequently occurring alphabet in the input sequence D and ' w ' is the number of words required to represent σ bits.

Section	Search Cost (λ) ($\lambda = \Omega(1) \wedge \lambda = O(\log n)$)	Data Structure	
		Space	Time
Sec. 7.2.1	$O(1)$	$O(n\sigma)$	$O(n\sigma)$
Sec. 7.2.2	$O(\log f_{\max})$	n	n
Sec. 7.2.4	$O(\lambda)$	$O(n + w2^{k\lambda})$	$O(kn + w2^{k\lambda})$
Sec. 7.2.5	$O(\lambda)$	$O(\min(n\sigma, n^2/(2^{k\lambda})))$	$O(\min(n\sigma, n^2/(2^{k\lambda})))$

Table 7.1: Comparison of the Different Search Methods

Chapter 8

Conclusions

In this thesis we have described how the contour based approach can be used effectively for solving the *LTS* problem. We have designed an algorithm for the same which performs better than the previous $O(n^2)$ algorithm (based on dynamic programming) suggested by Kosowski [8]. We have also described a new data structure for solving the *LCS* problem. In addition to this, we have discussed different search data structures that could be used in the *LCS* problem.

It has to be seen, if any of these search methods could be used to obtain a more efficient solution to the *LTS* problem. Another related problem in which these search techniques could be possibly used is that of finding the Longest Common Increasing Subsequence(*LIS*); Yang et. al [17] have proposed an algorithm with $O(mn)$ space and time complexities. Given two sequences $A[a_1a_2...a_m]$ and $B[b_1b_2...b_n]$ in which symbols are comparable with each other, the *LIS* is a longest possible common subsequence in which the symbols are in strictly increasing order.

Appendix A

The Algorithm(Pseudocode)

Algorithm 1 : Preprocessing

```
for  $a \in \Sigma$  do
   $construct(Next_a[0...(n-1)])$ 
end for
for  $i = 1$  to  $n/2$  do
   $Extend\_Contours(S_L[s_1...s_{i-1}], i, S_R[s_{(n/2+1)}...s_n])$ 
end for
 $p_{max} = p_{n/2}$ 
```

Algorithm 2 : $LTS(S[s_1s_2...s_n])$

```
for  $j = 1$  to  $p_{max}$  do
   $SHIFT\_RIGHT(S_L[s_1...s_{j-1}], S_R[s_j...s_n])$  {shift one symbol from  $S_R$  to  $S_L$ }
  if  $(p_{(n/2+j)} > p_{max})$  then
     $p_{max} \leftarrow p_{(n/2+j)}$ 
     $T \leftarrow LCS(S_L, S_R)$ 
  end if

   $SHIFT\_LEFT(S_L[s_1...s_j], S_R[s_{j+1}...s_n])$  {shift one symbol from  $S_L$  to  $S_R$ }
  if  $(p_{(n/2-j)} > p_{max})$  then
     $p_{max} \leftarrow p_{(n/2-j)}$ 
     $T \leftarrow LCS(S_L, S_R)$ 
  end if
end for

return  $T[t_1t_2...t_{p_{max}}]$ 
```

Algorithm 3 : *SHIFT_RIGHT*($S_L[s_1...s_{j-1}]$, $S_R[s_j...s_n]$)

{*Re-adjust Contours*}
 $y \leftarrow \text{find}((j+1), T_1)$
 $T_{\text{right}} \leftarrow \text{splay}(j, T_1)$
 $i \leftarrow T_{\text{right}} \rightarrow \text{row}$
 $x \leftarrow \text{Next}_{s_i}[j]$
 $T_{\text{temp}} \leftarrow \text{delete}(j, T_{\text{right}})$ {Delete column j }

for $k = 2$ to p_j do
 {insert new dominant match if it exists}
 if $x < y$ then
 $T_{\text{right}} \leftarrow \text{insert}(x, T_{\text{temp}})$
 else
 $T_{\text{right}} \leftarrow \text{splay}(y, T_{\text{temp}})$
 end if

$z \leftarrow T_{\text{right}} \rightarrow \text{key}$
 $T_{\text{left}} \leftarrow \text{splay}(z, T_k)$
 $i \leftarrow T_{\text{left}} \rightarrow \text{row}$
 $x \leftarrow \text{Next}_{s_i}[z]$
 $y \leftarrow \text{find}((T_{\text{left}} \rightarrow \text{key}) + 1, T_{\text{left}})$

$T_{\text{temp}} \leftarrow T_{\text{left}} \rightarrow \text{right}$
 $(T_{\text{left}} \rightarrow \text{right}) \leftarrow \text{NULL}$
 $T_{k-1} \leftarrow \text{join}(T_{\text{left}}, T_{\text{right}})$

end for

{*Extend Contours*}
 $i \leftarrow \text{Next}_{s_j}[j]$
 $k \leftarrow 1$
while $((T_k \neq \text{NULL}) \wedge (i \neq \text{NULL}))$ do
 if $(i < \text{First}_{T_k})$ then
 { First_{T_k} is the leftmost dominant match of the contour C_k }
 $T_k \leftarrow \text{insert}(i, T_k)$
 $i \leftarrow \text{Next}_{s_j}[\text{First}_{T_k}]$
 $k \leftarrow k + 1$
 else if $(i = \text{First}_{T_k})$ then
 $i \leftarrow \text{Next}_{s_j}[i]$
 $k \leftarrow k + 1$
 else
 $k \leftarrow k + 1$
 end if
end while
if $(i \neq \text{NULL})$ then
 $T_{k+1} \leftarrow \text{create}((i, j))$ {Create a new tree}
 $p_{j+1} \leftarrow (k + 1)$
end if

Algorithm 4 : *SHIFT_LEFT*($S_L[s_1 \dots s_j]$, $S_R[s_{j+1} \dots s_n]$)

```
for  $k = 1$  to  $k = p_j$  do
  if  $(j = \text{First}_{T_k} \rightarrow \text{row})$  then
     $T_k \leftarrow \text{delete}(\text{row}, T_k)$ 
  end if
end for
 $x \leftarrow \text{Next}_{s_j}[1]$ 
 $T_{\text{temp}} \leftarrow \text{insert}(x, T_{\text{temp}})$ 

for  $k = 1$  to  $k = p_j$  do
   $T_k \leftarrow \text{splay}(x, T_k)$ 
   $T_{\text{left}} \leftarrow T_k \rightarrow \text{left}$ 
   $(T_k \rightarrow \text{left}) \leftarrow \text{NULL}$ 
  if  $x = T_k \rightarrow \text{row}$  then
     $T_k \rightarrow \text{key} \leftarrow j$ 
  end if
   $T_k \leftarrow \text{join}(T_{\text{temp}}, T_k)$ 
   $x \leftarrow T_{\text{left}} \rightarrow \text{row}$ 
   $T_{\text{temp}} \leftarrow T_{\text{left}}$ 
end for

if  $T_{\text{temp}} \neq \text{NULL}$  then
   $p_{(j-1)} \leftarrow p_j + 1$ 
   $T_{p_{(j-1)}} \leftarrow T_{\text{temp}}$ 
end if
```

Algorithm 5 : *LCS*($S_L[s_1 s_2 \dots s_l]$, $S_R[s_{l+1} \dots s_n]$)

```
 $\text{match} \leftarrow T_{p_l} \rightarrow \text{key}$ 
 $T[p_l] \leftarrow S_R[\text{match}]$ 
for  $k = (p_l - 1)$  to  $1$  do
  {Find a predecessor match  $(i', j') \in C_k$  for  $(i, j) \in C_{k+1}$  such that  $(i' < i) \wedge (j' < j)$ }
   $\text{match} \leftarrow \text{find}(\text{match}, T_k)$ 
   $T[k] = S_R[\text{match}]$ 
end for
return  $T[t_1 t_2 \dots t_{p_l}]$ 
```

Appendix B

Splay Trees

Splay trees were developed by Sleator and Tarjan [16]. There are a kind of self organizing binary search tree, in which search operation can be performed in $O(\log n)$ amortized time. The following are some of the operations that can be performed on a splay tree, (as presented in Mehlhorn [10]):

Access(x, T): If element x in the tree T , then return a pointer to its location, otherwise return *NULL*.

Insert(x, T): Insert x into the tree T and return the pointer to the root.

Delete(x, T): Delete x from tree T and return the resulting tree.

Join(T_1, T_2): Return a tree representing the elements of T_1 followed by the elements of T_2 , destroying T_1 and T_2 . (This assumes all the elements of T_1 are smaller than all the elements of T_2).

Split(x, T): Returns two trees T_1 and T_2 ; T_1 contains all the elements of T smaller than x and T_2 contains all the elements of T larger than x (this assumes that x is in the tree); tree T is destroyed.

The main operation of the splay tree is the *Splay*(x, T) operation, using which all the above operations can be performed.

Splay(x, T): Returns a tree representing the same set of elements as T . If x is in the tree then x becomes the root, otherwise, either the immediate predecessor of x or the immediate successor of x in T becomes the root. This operation destroys T .

Appendix C

Comparison Results

		Rick[13]'s Algorithm		Our Algorithm	
$n = D $	$m = Q $	Preprocessing Time	Overall Time	Preprocessing Time	Overall Time
$\sigma = 40$					
200	25	190	196	89	158
400	50	382	413	115	379
500	80	496	543	125	626
700	125	708	795	140	1327
$\sigma = 50$					
300	75	390	412	112	396
500	100	643	686	132	702
700	150	914	1013	156	1382
1000	250	1486	1787	168	3535
$\sigma = 65$					
400	100	696	770	138	541
500	125	867	923	151	775
700	175	1225	1367	172	1332
1000	300	1838	2155	188	2989
$\sigma = 80$					
500	150	1095	1169	159	799
700	225	1592	1720	182	1504
1500	400	3383	4013	244	5231
1750	450	3813	4651	261	6483
$\sigma = 94$					
500	150	1270	1345	168	694
700	250	1931	2111	198	1551
1500	300	3728	4197	262	3253
2000	400	4970	5800	288	5765
3000	500	7298	8793	351	10128

Table C.1: Comparative Analysis of *LCS* algorithms($\sigma \geq (n/\log^2 n)$)(Time in μs)

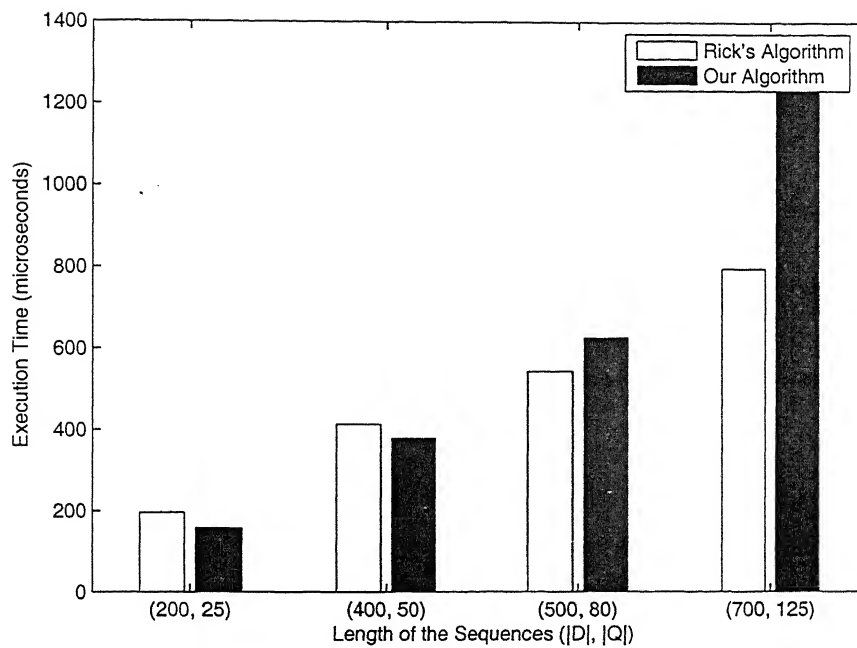


Figure C.1: $\sigma = 40$

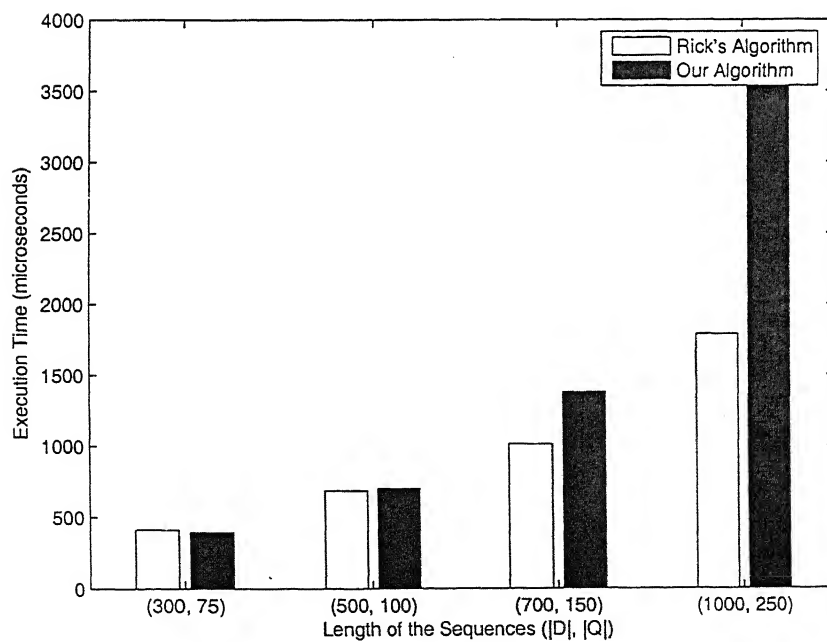


Figure C.2: $\sigma = 50$

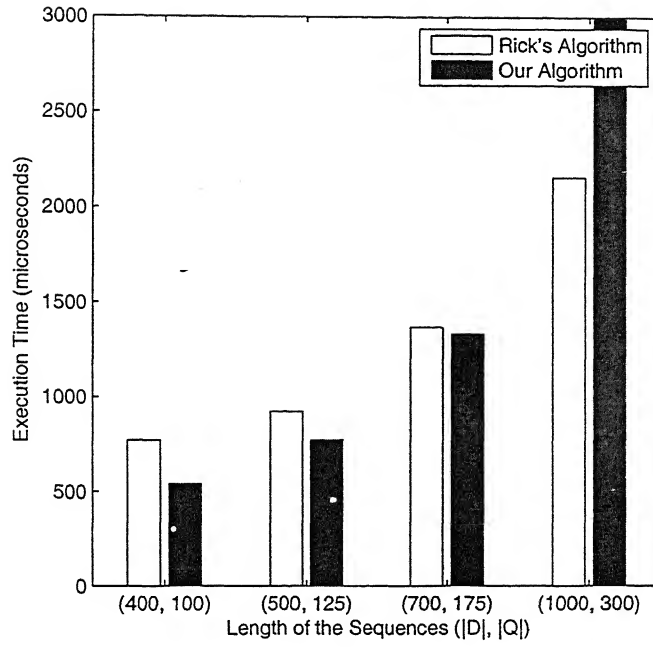


Figure C.3: $\sigma = 65$

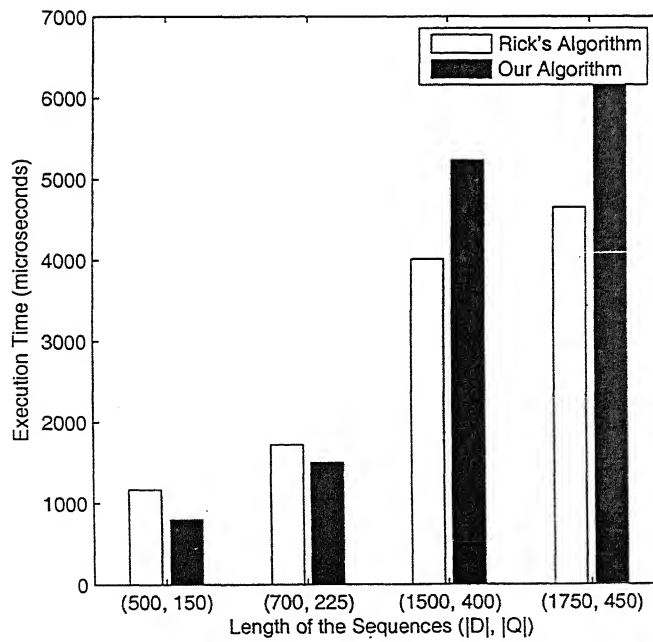


Figure C.4: $\sigma = 80$

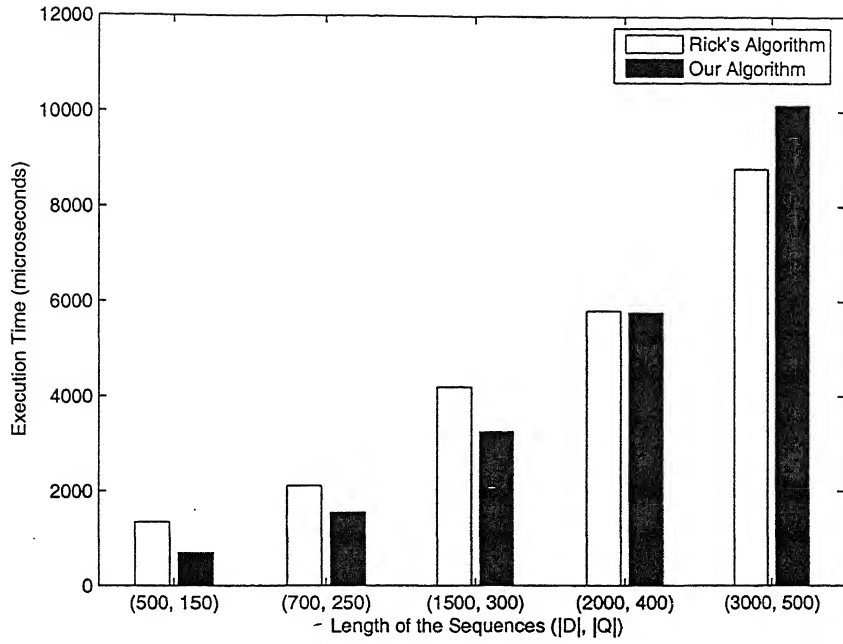


Figure C.5: $\sigma = 94$

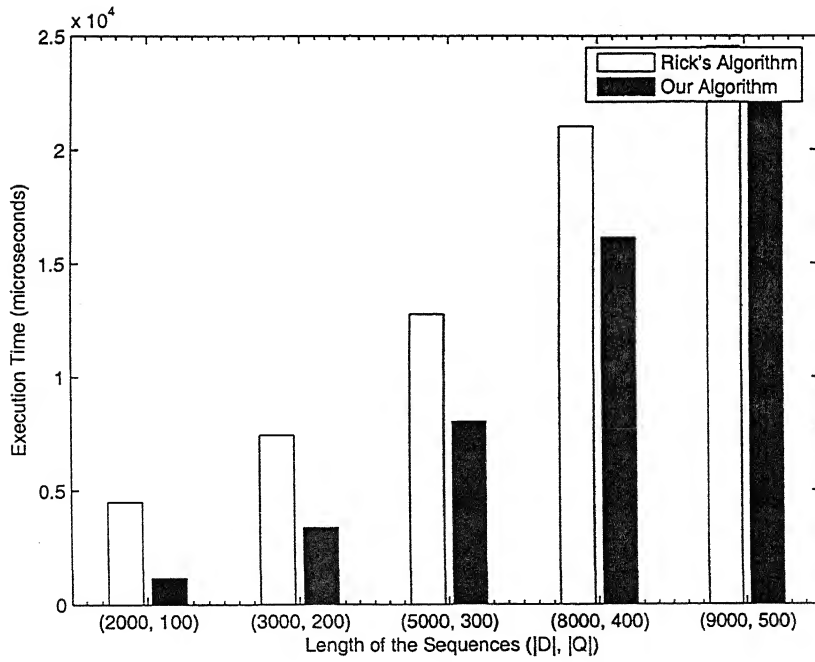


Figure C.6: $\sigma = 94$ and $n \gg m$

Bibliography

- [1] BERGROTH, L., HAKONEN, H., AND RAITA, T. A survey of longest common subsequence algorithms. In *SPIRE* (2000), pp. 39–48.
- [2] BERGROTH, L., HARRI, H., AND VAISANEN, J. New refinement techniques for longest common subsequence algorithms. In *String Processing and Information Retrieval, Proceedings* (2003), pp. 287–303.
- [3] CHATTARAJ, A., AND PARIDA, L. An inexact-suffix-tree-based algorithm for detecting extensible patterns. *Theor. Comput. Sci.* **335**, 1 (2005), 3–14.
- [4] HIRSCHBERG, D. S. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM* **18**, 6 (1975), 341–343.
- [5] HIRSCHBERG, D. S. Algorithms for the longest common subsequence problem. *J. ACM* **24**, 4 (1977), 664–675.
- [6] HSU, W. J., AND DU, M. W. New algorithms for the lcs problem. *J. Comput. Syst. Sci.* **29**, 2 (1984), 133–152.
- [7] KOLPAKOV, R., AND KUCHEROV, G. Finding maximal repetitions in a word in linear time. In *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science* (1999), pp. 596–604.
- [8] KOSOWSKI, A. An efficient algorithm for the longest tandem scattered subsequence problem. In *SPIRE* (2004), pp. 93–100.
- [9] MAIN, M. G., AND LORENTZ, R. J. An $o(n \log n)$ algorithm for finding all repetitions in a string. *Journal of Algorithms* **5**, 3 (1984), 422–432.

- [10] MEHLHORN, K. *Data structures and algorithms 1: sorting and searching*. Springer-Verlag New York, Inc., New York, NY, USA, 1984.
- [11] MEHLHORN, K. *Data structures and algorithms 3: multi-dimensional searching and computational geometry*. Springer-Verlag New York, Inc., New York, NY, USA, 1984.
- [12] NAKATSU, N., KAMBAYASHI, Y., AND YAJIMA, S. A longest common subsequence algorithm suitable for similar text strings. *Acta Inf.* 18 (1982), 171–179.
- [13] RICK, C. A new flexible algorithm for the longest common subsequence problem. In *Proc. CPM'95, Lecture Notes in Computer Science* (1995), vol. 937, Springer, pp. 340–351.
- [14] RICK, C. Simple and fast linear space computation of longest common subsequences. *Inf. Process. Lett.* 75, 6 (2000), 275–281.
- [15] SKIENA, S. S. *The Algorithm Design Manual*. Springer-Verlag New York, Inc., 1998.
- [16] SLEATOR, D. D., AND TARJAN, R. E. Self-adjusting binary search trees. *J. ACM* 32, 3 (1985), 652–686.
- [17] YANG, I.-H., HUANG, C.-P., AND CHAO, K.-M. A fast algorithm for computing a longest common increasing subsequence. *Inf. Process. Lett.* 93, 5 (2005), 249–253.